

Programação Orientada a Objetos

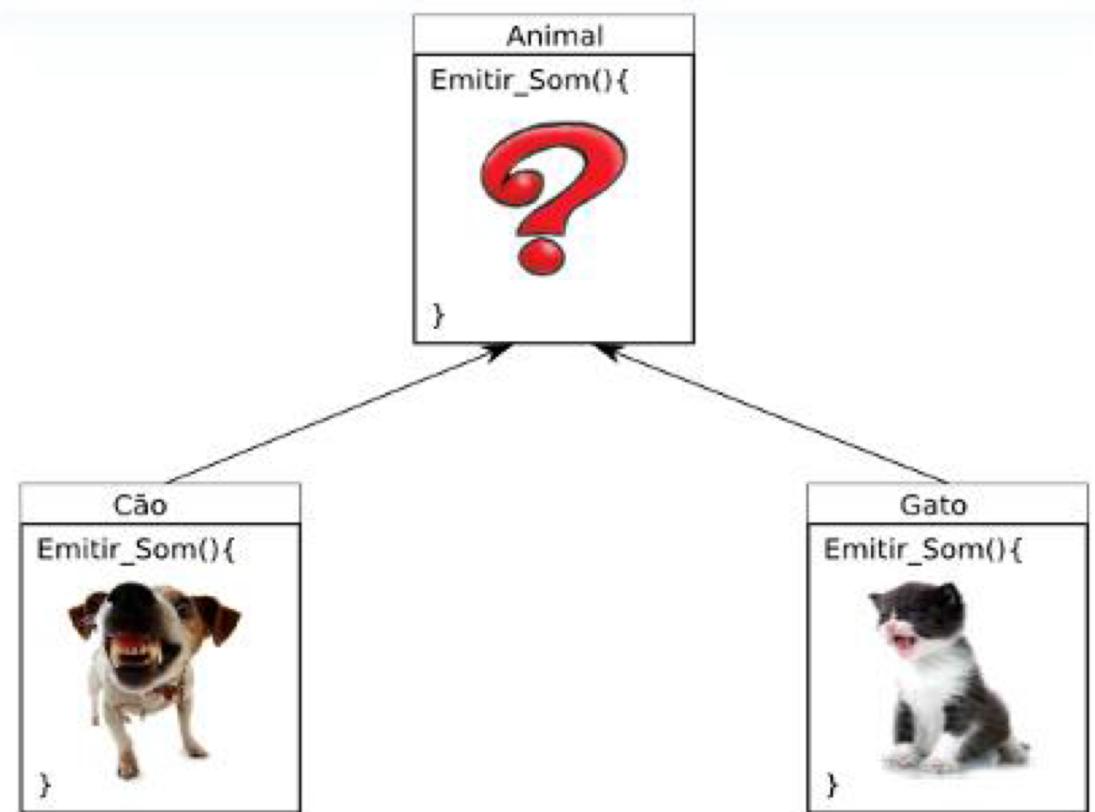
Nádia Félix

nadia.felix@ufg.br

Programação Orientada a Objetos

POLIMORFISMO

Muitas formas



Programação Orientada a Objetos

Polimorfismo

Permite a um mesmo objeto se manifestar de diferentes formas;

Permite que uma mesma operação possa ser definida para diferentes tipos de classes, e cada uma delas a implementa como quiser;

Programação Orientada a Objetos

Polimorfismo

É o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem se comportar de forma diferente;

Invocar métodos que têm a mesma assinatura, mas com comportamentos distintos ... especializados para cada subclasse.

O polimorfismo permite “programar no geral” em vez de “programar no específico”.

Programação Orientada a Objetos

Polimorfismo em prática

```
public class Animal {  
    private String nome;  
    private int coordenadaX;  
    private int coordenadaY;  
  
    public Animal(String nome) {  
        this.nome = nome;  
    }  
  
    public Animal() {  
        this.nome = "anonimo";  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    protected void setCoordenadas(int x, int y) {  
        coordenadaX = x;  
        coordenadaY = y;  
    }  
  
    public void mover(int x, int y) {  
        System.out.println("Não sei me mover");  
    }  
}
```

```
class Anfibio extends Animal {  
    public Anfibio(String nome) {  
        super(nome);  
    }  
  
    public void mover(int x, int y) {  
        setCoordenadas(x, y);  
        System.out.println("Movimento do Anfibio");  
    }  
}  
  
class Ave extends Animal {  
    public Ave(String nome) {  
        super(nome);  
    }  
  
    public void mover(int x, int y) {  
        setCoordenadas(x, y);  
        System.out.println("Movimento da Ave");  
    }  
}  
  
class Peixe extends Animal {  
    public Peixe(String nome) {  
        super(nome);  
    }  
  
    public void mover(int x, int y) {  
        setCoordenadas(x, y);  
        System.out.println("Movimento do Peixe");  
    }  
}
```

Programação Orientada a Objetos

Polimorfismo em prática (cont..)

```
public class MundoAnimal {  
    public static void main (String args[]) {  
        Animal reino[];  
        reino = new Animal[3];  
  
        reino[0] = new Anfibio("Salamandra");  
        reino[1] = new Ave("Quero-quero");  
        reino[2] = new Peixe("Dourado");  
  
        for (int i=0; i<3; i++) {  
            reino[i].mover(10, 10);  
        }  
    }  
}
```

- Temos um vetor de Animais.
- Os elementos do vetor são instanciados com subclasses de animais.
- O método mover() é chamado para cada elemento do vetor.

Qual método mover() vai ser chamado? O método da classe Animal ou os métodos especializados de cada uma das subclasses?

Programação Orientada a Objetos

Polimorfismo em prática (cont..)

A decisão sobre qual método sobrescrito deve ser selecionado (dentro da hierarquia de classes) é tomada em tempo de execução, considerando a classe da instância (objeto) que o está chamando.

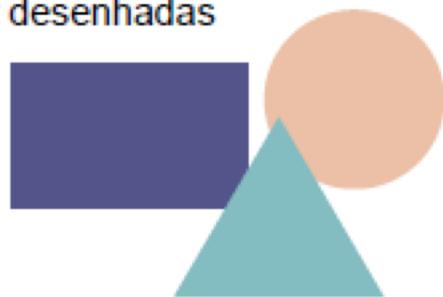
O polimorfismo permite codificar programas que processam objetos que compartilham a mesma superclasse como se todos eles fossem objetos daquela superclasse (simplificando a programação).

Programação Orientada a Objetos

Polimorfismo em prática (cont..)

Com o polimorfismo, podemos projetar e implementar sistemas facilmente extensíveis ... novas classes podem ser adicionadas com pouca ou nenhuma modificação nas partes gerais do programa ... desde que as novas classes façam parte da hierarquia de herança já existente.

Retângulo, círculo e triângulo são formas geométricas que podem ser desenhadas



Com o polimorfismo o método `desenhar()` de cada subclasse pode ser sobreescrito para que se comporte de maneira diferente.

Um programa que faz desenhos e que utiliza formas geométricas não precisa se preocupar em chamar métodos específicos de cada forma ... basta chamar o método `desenhar()` da hierarquia ... se a subclasse for um círculo ... seu método específico será chamado.



Programação Orientada a Objetos

Casting ou Moldagem

Utilizado para converter um objeto ou tipo primitivo de um tipo/classe para outro;

Suponhamos a necessidade de armazenar um valor `int` dentro de um `double`... como a precisão de um `double` é maior ... a conversão é natural ...

```
int i = 3;
```

```
double pi= i + .14159;
```

Quando há perda de precisão ... O *casting* é necessário ...

```
double pi= 3.14159;
```

```
int i = (int) pi; // Ao final i vale 3
```

Programação Orientada a Objetos

Casting de Objetos

Em aplicações que exploram o polimorfismo é comum a necessidade de fazer com que um objeto se passe por outro.

O *casting* não modifica o objeto, é o receptor do *cast* que constitui um novo objeto ou um novo tipo ...

Ex:

```
Anfibio sapo = new Anfibio("Sapo Boi");
```

```
Animal animal = (Animal) sapo;
```

Programação Orientada a Objetos

Casting de Objetos

```
public class Teste {  
  
    public static void apresentar(Animal a) {  
        System.out.println( a.getNome() );  
        a.mover(10,10);  
    }  
  
    public static void main (String args[]) {  
  
        Object lista[];  
  
        lista = new Object[3];  
  
        lista[0] = new Anfibio("Salamandra");  
        lista[1] = new Ave("Quero-quero");  
        lista[2] = new Peixe("Dourado");  
  
        for (int i=0; i<3; i++) {  
            apresentar( (Animal) lista[i]);  
        }  
    }  
}
```

Temos um vetor de Objetos.

Um método que recebe um animal, mostra seu nome e o movimenta

Saída do código

```
Salamandra  
Movimento do Anfíbio  
  
Quero-quero  
Movimento da Ave  
  
Dourado  
Movimento do Peixe
```

Antes de chamar o método,
os objetos são transformados
em Animais ...

Programação Orientada a Objetos

Operador `instanceof`

Utilizado para determinar o tipo de um objeto em tempo de execução;

Utilizado em situações onde alguma operação específica de uma subclasse precisa ser chamada, mas antes é necessário verificar se o objeto que vai chamar o método é do tipo correto.

Utilização: `refObjeto instanceof nomeClasse`

Retorna um valor booleano (true ou false) indicando se o objeto referenciado (refObjeto) é realmente uma instância da classe (nomeClasse)

Programação Orientada a Objetos

Instanceof em Prática

```
public class MundoAnimal {  
  
    public static void main (String args[]) {  
  
        Animal reino[];  
        reino = new Animal[3];  
  
        reino[0] = new Anfibio("Salamandra");  
        reino[1] = new Ave("Quero-quero");  
        reino[2] = new Peixe("Dourado");  
  
        for (int i=0; i<3; i++) {  
  
            if (reino[i] instanceof Peixe) {  
                ((Peixe) reino[i]).nadar();  
            }  
            else {  
                reino[i].mover(10,10);  
            }  
  
        }  
    }  
}
```

Se a instância for da classe Peixe, então chama o método nadar (o casting é necessário para evitar um erro de compilação).

Programação Orientada a Objetos

Herança e Polimorfismo na prática

```
public class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
  
    public double bonificacao(){  
        double b = salario * 0.10;  
        return b;  
    }  
  
    public double getSalario() {  
        return salario;  
    }  
  
    public void setSalario(double salario) {  
        this.salario = salario;  
    }  
}
```

```
public class Gerente extends Funcionario{  
    private int senha;  
  
    public double bonificacao(){  
        double b = salario * 0.15;  
        return b;  
    }  
}
```

```
public class TestaHeranca {  
  
    public static void main(String [] args){  
        Gerente g = new Gerente();  
        g.setSalario(3000);  
        System.out.println("A bonificacao é:  
            " + g.bonificacao());  
    }  
}
```

Programação Orientada a Objetos

Herança e Polimorfismo na prática

Na herança, vimos que Gerente é um Funcionário, pois é uma extensão deste.

Podemos referenciar a um Gerente como sendo um Funcionário, ou seja, podemos tratar Gerente como Gerente ou apenas como um Funcionário.

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas

CUIDADO: polimorfismo não quer dizer que o objeto fica se transformando. Muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que muda é a maneira como nos referenciamos a ele.

Programação Orientada a Objetos

Herança e Polimorfismo na prática

Qual o retorno do código abaixo, 300 ou 450?

```
public class TestarHeranca2 {  
  
    public static void main(String[] args) {  
        Funcionario f = new Gerente("Marcia","987654321",3344);  
        f.setSalario(3000);  
        System.out.println("Nome Gerente: " + f.getNome());  
        System.out.println("Bonificacao: " + f.bonificacao() );  
    }  
}
```

450 pois o tipo de dados que criamos foi Gerente que tem a bonificação de 15%

Programação Orientada a Objetos

Herança e Polimorfismo na prática

Mas em que situação isso seria útil?

Por exemplo, no caso de precisarmos passar um Funcionário como parâmetro de um método:

```
public class ControleDeGastos {  
  
    private double gastosComBonificacao = 0;  
  
    public void bonificacaoPorFuncionario(Funcionario f){  
        gastosComBonificacao += f.bonificacao();  
    }  
    public double getGastosComBonificacao(){  
        return gastosComBonificacao;  
    }  
}
```

Programação Orientada a Objetos

Herança e Polimorfismo na prática

Mas em que situação isso seria útil?

Por exemplo, no caso de precisarmos passar um Funcionário como parâmetro de um método:

```
public class TestaControleDeGastos {  
    public static void main(String[] args) {  
        ControleDeGastos cg = new ControleDeGastos();  
        Funcionario f = new Funcionario("Jose", "123456789");  
        f.setSalario(2000);  
        Gerente g = new Gerente("Maria", "33456787", 123456);  
        g.setSalario(3000);  
        cg.bonificacaoPorFuncionario(f);  
        System.out.println("Gastos Parciais: " + cg.getGastosComBonificacao());  
        cg.bonificacaoPorFuncionario(g);  
        System.out.println("Gastos Parciais: " + cg.getGastosComBonificacao());  
    }  
}
```

Programação Orientada a Objetos

```
public class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
  
    Funcionario(String nome, String cpf){  
        This.nome = nome;  
        This.cpf = cpf;  
    }  
    ...  
}
```

```
public class Gerente extends Funcionario{  
    private int senha;  
  
    Gerente (String nome, String cpf, int senha){  
        super(nome, cpf);  
        This.senha = senha;  
    }  
    public void limpaSenha() {  
        senha = 123456;  
    }  
}
```

```
public class TestaPolimorfismo {  
    public static void main(String[] args) {  
        Funcionario[] lista = new Funcionario[10];  
        lista[0] = new Funcionario("Maria", "123456789");  
        lista[1] = new Gerente("Paulo", "987654321", 1234);  
        ...  
        for(int i=0; i<lista.length; i++) {  
            if (lista[i] instanceof Gerente) {  
                ((Gerente) lista[i]).limpaSenha();  
            }  
        }  
    }  
}
```